EECS 280 – Lecture 20

Functors, Function Pointers, and Impostor Syndrome

1

2 Review: Traversal by Iterator

- Walk an **iterator** across the elements.
- To get an element, just dereference the iterator!
- Get iterators that define the range from the container using `begin()` and `end()` functions.

```
List<int> list;
int arr[3] = { 1, 2, 3 };
fillFromArray(list, arr, 3);

List<int>::Iterator end = list.end();
for (List<int>::Iterator it = list.begin(); it != end; ++it) {
  cout << *it << endl;
}
```

Ask the List for iterators that define the sequence of elements.

3 Review: The Iterator Interface

- Iterators provide a common interface for iteration.
  - A generalized version of traversal by pointer.
  - An iterator "points" to an element in a container and can be "incremented" to move to the next element.
- Iterators[1] support these operations:
  - Dereference – access the current element. `*it`
  - Increment – move forward to the next element. `++it`
  - Equality – check if two iterators point to the same place. `it1 == it2` `it1 != it2`

1 There are many different kinds of iterators. These operations are specifically required for *input iterators*.

6/9/2022

4 Review: List Iterator

```
template <typename T>
class List {
public:
  ...
  class Iterator {
    friend class List;
  public:
    Iterator() : node_ptr(nullptr) { }
    T & operator*() const;
    Iterator & operator++();
    bool operator==(Iterator rhs) const;
    bool operator!=(Iterator rhs) const;
  private:
    Iterator(Node *np) : node_ptr(np) { }
    Node *node_ptr;
  };

  Iterator begin() { return Iterator(first); }
  Iterator end() { return Iterator(); }
  ...
};
```

fancy pointer

5 Example: any_of_odd

- Implement this function that checks if there are any odd elements in a sequence.

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Returns true if any element in the
//           sequence [begin, end) is odd.
template <typename Iter_type>
bool any_of_odd(Iter_type begin, Iter_type end) {
  for (Iter_type it = begin; it != end; ++it) {
    if (*it % 2 != 0) { return true; }
  }
  return false;
}
```

```
int main() {
  List<int> list; // Fill with numbers
  cout << any_of_odd(list.begin(), list.end());
}
```

We could use some other container here too!

Alternate Solutions: any_of_odd

6

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Returns true if any element in the
//           sequence [begin, end) is odd.
template <typename Iter_type>
bool any_of_odd(Iter_type begin, Iter_type end) {
  while (begin != end) {
    if (*begin % 2 != 0) { return true; }
    ++begin;
  }
  return false;
}
```

其他的处理方式

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Returns true if any element in the
//           sequence [begin, end) is odd.
template <typename Iter_type>
bool any_of_odd(Iter_type begin, Iter_type end) {
  for ( ; begin != end; ++begin) {
    if (*begin % 2 != 0) { return true; }
  }
  return false;
}
```

6/9/2022

7 A General any_of Function

- If we were to write an `any_of_even` function, it would look very similar.
  - Code duplication is bad!
- The only piece of code that needs to change is the test for oddness/evenness.

```
if (*it % 2 != 0) { return true; }
```

predicate

为了避免duplication，我们希望每次之更改predicate。这个条件可以是偶数，奇数等等。

6/9/2022

8 Reducing Code Duplication

Bad

```
int times2(int x) {
  return x * 2;
}

int times3(int x) {
  return x * 3;
}

int times4(int x) {
  return x * 4;
}

int main() {
  cout << times2(42);
  cout << times3(42);
  cout << times4(42);
}
```

Good

```
int times(int x, int n) {
  return x * n;
}

int main() {
  times(42, 2);
  times(42, 3);
  times(42, 4);

  for (int i = 0; i < 10; ++i) {
    cout << times(42, i);
  }

}
```

6/9/2022

9 Idea: Function Parameters

```
bool is_even(int x);
bool is_odd(int x);
bool is_prime(int x);

template <typename Iter_type>
bool any_of(Iter_type begin, Iter_type end, ____ fn) {

  for (Iter_type it = begin; it != end; ++it) {
    if ( fn(*it) ) { return true; }
  }

  return false;
}
```

predicate在这里输入

A parameter that takes a function!

The function tells us "yes" or "no" for each item.

```
int main() {
  List<int> list; // Fill with numbers
  cout << any_of(list.begin(), list.end(), is_prime);
}
```

Pass in the function we want to use.

6/9/2022

10 Function Pointers

- Variables that can store a function.
- The instructions for executing a function are stored somewhere…a function pointer actually stores the **address** where the function is stored.

我们不能直接pass function但是我们可以pass进去function的pointer。

higher-order-function的意思是一个function的parameter中是另一个function。

6/9/2022

10 Function Pointers

- Variables that can store a function.
- The instructions for executing a function are stored somewhere…a function pointer actually stores the **address** where the function is stored.
- Unlike a regular pointer, you don't need to dereference a function pointer to use it.
- The syntax for declaring a function pointer looks a lot like the functions it can hold:

In a declaration, the * means "a pointer to". Parentheses are needed due to precedence.

```
int max(int x, int y) { return x > y ? x : y; }
int min(int x, int y) { return x < y ? x : y; }

int (*fn)(int, int) = max; // fn starts as max
fn = min; // set fn to be min
cout << fn(2, 5); // uses min, prints 2
```

Takes two ints.

Returns an int.

6/9/2022

11 Declaration Syntax

- C++ declarations are read "inside out".
- Postfix operators ([], ()) have higher precedence than prefix (*, &).
- Example: `double (*func)(int)`

`double (*func)(int)`	func is
`double (*func)(int)`	a pointer to
`double (*func)(int)`	a function
`double (*func)(int)`	that takes an int
`double (*func)(int)`	and returns a double

12 Function Pointers for any_of

- Basically, "A variable that holds a function."

```
bool is_odd(int x) {
  return x % 2 != 0;
}
```

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Returns true if there is an element in
//           [begin, end) for which fn returns true.
template <typename Iter_type>
bool any_of(Iter_type begin, Iter_type end,
            bool (*fn)(int)) {
  for (Iter_type it = begin; it != end; ++it) {
    if (fn(*it)) { return true; }
  }
  return false;
}
```

"fn is a pointer to a function that takes an int and returns a bool."

```
int main() {
  List<int> list; // Fill with numbers
  cout << any_of(list.begin(), list.end(), is_odd);
}
```

The type of is_odd matches the type of the fn parameter.

13 Predicates

- A **predicate** is a function that returns either `true` or `false` depending on whether its argument has some property.
  - `is_odd`, `is_even`, `is_prime`, etc.
- Thus, a better name for the third parameter to `any_of` might be `pred`.

```
bool any_of(Iter_type begin, Iter_type end,
            bool (*pred)(int));
```

- The strategy here is to write a general function like `any_of` and customize its behavior with a predicate.

14 Making Predicates

- What if we wanted to use `any_of` to check for elements greater than some threshold?
  - Greater than 0?

```
bool greater0(int x) {
  return x > 0;
}
```

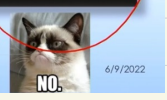
  - Greater than 32?

```
bool greater32(int x) {
  return x > 32;
}
```

  - Greater than 212?

```
bool greater212(int x) {
  return x > 212;
}
```

  - Is this a good approach?



问题在于当用户输入一个温度值T时，我们不知道如何分辨调用其中哪个方程。

这个问题有解决方法。

6/9/2022

解决思路，但是问题是function没有private variable，所以我们用class来解决这个问题。

15 Exercise

```
int main() {
  List<int> list; // Fill with numbers
  cout << any_of(list.begin(), list.end(), is_odd);
}
```

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Returns true if there is an element in
//           [begin, end) for which fn returns true.
template <typename Iter_type>
bool any_of(Iter_type begin, Iter_type end,
            bool (*fn)(int)) {
  for (Iter_type it = begin; it != end; ++it) {
    if (fn(*it)) { return true; }
  }
  return false;
}
```

```
bool is_odd(int x) {
  return x % 2 != 0;
}

bool greater32(int x) {
  return x > 32;
}

bool greater212(int x) {
  return x > 212;
}
```

Question 19.1

What can we do to fix the problem?

A) Make a single greater function that uses a global variable to store the threshold.
B) Make a single greater function with an extra parameter to pass in the threshold.
C) Add an extra parameter to the any_of function to pass in the threshold.
D) These are all good ideas.
E) None of these are good ideas.

16 Functions and First-Class Objects

- A **first-class object**…
  - …has state. (i.e. stores some information)
  - …can be created at runtime.
  - …can be passed as an argument.
  - …etc.
- C++ does not have first-class functions. This gives us two big limitations:
  - Functions can't really store information.
  - You can't just make a function at runtime.

17 Functors

- We would like to be able to make functions on the fly, but we can't.
- Instead, we'll make class-type objects that pretend to be functions.
  - We'll overload the **function call operator**.
  - Yes, you can do this!
- An **iterator** is an object that acts like a **pointer**.
- A **functor** is an object that acts like a **function**.

Slides

在这种情况下，我们不能使用function pointer，因为我们需要在run time从user手中接受一个数值，并将这个数值赋值给threshold。但是function pointer需要在compile time就指向一个已经写好的function，不能在run time对其进行修改。

18 Implementing a GreaterN Functor

```
class GreaterN {
private:
  int threshold;

public:
  GreaterN(int threshold_in)
    : threshold(threshold_in) { }

  bool operator()(int n) const {
    return n > threshold;
  }
};
```

A GreaterN object stores a threshold as a member variable.

The threshold is passed into the constructor.

The overloaded function call operator takes an int and tests it.

```
int main() {
  GreaterN g0(0);
  GreaterN g32(32);
  GreaterN g212(212);

  cout << g0(-5); // false
  cout << g0(3);  // true

  cout << g32(9); // false
  cout << g32(45); // true
}
```

```
int main() {
  List<int> list; // Fill with numbers
  GreaterN g0(0);
  cout << any_of(list.begin(), list.end(), g0);
  cout << any_of(list.begin(), list.end(), GreaterN(32));
}
```

See file L19.1_GreaterN at lobster.eecs.umich.edu.

6/13/2022

19 Functors as Parameters

- To pass a functor into `any_of`, we add a template parameter for the type of the functor.

使用template来接受各种类型的function pointer和 functor来完成不同的任务

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Returns true if there is an element in
//           [begin, end) for which pred returns true.
template <typename Iter_type, typename Predicate>
bool any_of(Iter_type begin, Iter_type end,
            Predicate pred) {
  for (Iter_type it = begin; it != end; ++it) {
    if (pred(*it)) { return true; }
  }
  return false;
}
```

```
int main() {
  List<int> list; // Fill with numbers
  GreaterN g0(0);
  cout << any_of(list.begin(), list.end(), is_odd);
  cout << any_of(list.begin(), list.end(), g0);
  cout << any_of(list.begin(), list.end(), GreaterN(32));
}
```

We can still use any_of with a function pointer.

28 Recall: max_element

```
template <typename Iter_type>
Iter_type max_element(Iter_type begin, Iter_type end) {

  Iter_type maxIt = begin;

  for (Iter_type it = begin; it != end; ++it) {
    if (*maxIt < *it) {
      maxIt = it;
    }
  }
  return maxIt;
}
```

Start by assuming first element is the max.

Use traversal by iterator to check each element.

If we find a larger element, update maxIt to point to it.

Which element types can this function be used with? Those that support the < operator.

```
int main() {
  vector<int> vec; // fill with numbers
  cout << *max_element(vec.begin(), vec.end()) << endl;
}
```

Dereference returned iterator to get the element itself.

29 max_element with Ducks

- Using `max_element` with a container of `Ducks` doesn't work because there is no overloaded `<` operator for `Duck`.
- But there are still situations where we'd like to find the "maximum" `Duck`.
  - The `Duck` whose name is last alphabetically.
  - The `Duck` with the most ducklings.
- Here's the idea:

A functor that compares two Ducks according to their names.

```
int main() {
  vector<Duck> vec; // fill with Ducks
  max_element(vec.begin(), vec.end(), DuckNameLess());
}
```

30 Comparators

- A **comparator** is a function that compares two arguments and returns true/false depending on their ordering.
- It's common to use "less" comparators that return true if the first argument is less than the second argument.
  - `DuckNameLess`, `DuckDucklingsLess`, etc.

```
class DuckNameLess {
public:
  bool operator()(const Duck &d1, const Duck &d2) const {
    return d1.getName() < d2.getName();
  }
};
```

31 max_element with Ducks

```
template <typename Iter_type, typename Comparator>
Iter_type max_element(Iter_type begin, Iter_type end,
                      Comparator less) {
  Iter_type maxIt = begin;
  for (Iter_type it = begin; it != end; ++it) {
    if (less(*maxIt,*it)) {
      maxIt = it;
    }
  }
  return maxIt;
}
```

Call comparator rather than using < operator.

```
int main() {
  vector<Duck> vec; // fill with Ducks
  max_element(vec.begin(), vec.end(), DuckNameLess());
}
```

32 for_each

- Many different functions iterate over a sequence and do something with each element.
  - Functors serve as an abstraction for "do something".
  - Iterators serve as an abstraction for a sequence.
- Here's code for `for_each`, which follows this idea:

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS:  Applies func to each of the elements in
//           the sequence [begin, end) and returns func.
template <typename Iter_t, typename Func_t>
Func_t for_each(Iter_t begin, Iter_t end, Func_t func) {
  for (Iter_t it = begin; it != end; ++it) {
    func(*it);
  }
  return func;
}
```

We return the functor in case it contains some information about the result.

6/13/2022

33 Printing with for_each

```
template <typename T>
class Printer {
public:
  void operator()(const T &n) const {
    cout << n;
  }
};

template <typename Iter_t, typename Func_t>
Func_t for_each(Iter_t begin, Iter_t end, Func_t func) {
  for (Iter_t it = begin; it != end; ++it) {
    func(*it);
  }
  return func;
}

int main() {
  List<int> list; // Fill with numbers
  for_each(list.begin(), list.end(), Printer<int>());
}
```

Functor

maybe we can make a functor: multiplier (5);

return Adder instance.

we can write: Adder result = for_each(list.begin(), list.end(), Adder());

result.get_sum();

or a functor called : Adder

Then it will return the functor that it's been operating on.

```
class Adder {
private:
    int sum;
public:
    Adder() : sum(0) {}
    void operator()(int value) {
        sum += value;
    }
```